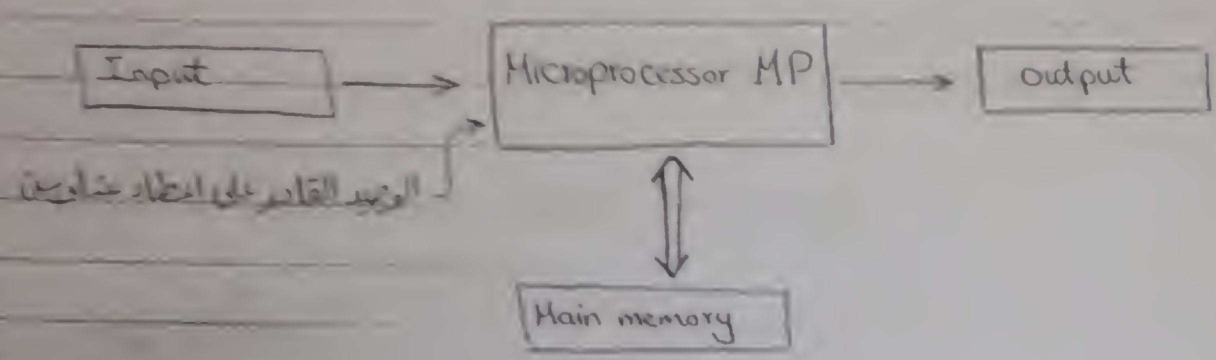
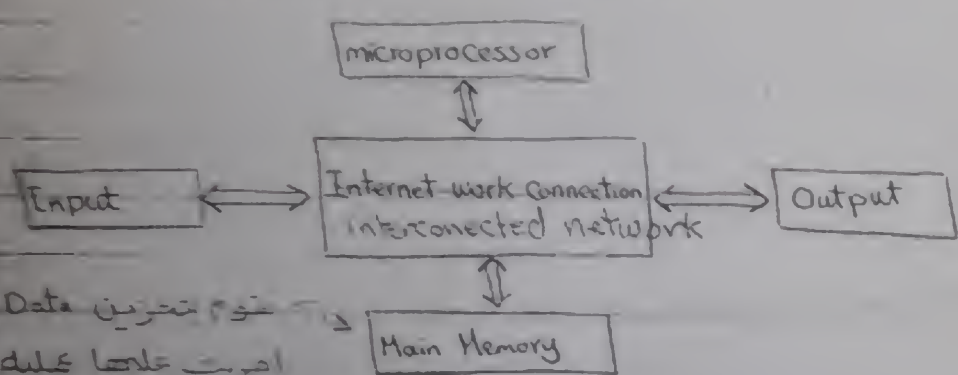
Ch 1 Microprocessor (Introduction)





تقوم بتخزين Data و programs
اجريت عليها المعالجة

Control processing Unit (cpu) ≡ Machine → microprocessor

→ cpu components

[1] ALU

[2] CU (control unit) تتحكم في وظائف الجهاز

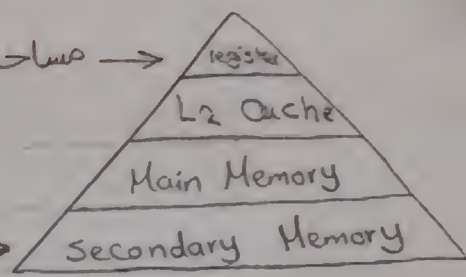
[3] Registers level 1 cache

→ cpu functions:

1- Execode arithmatic & logic instruction
(Fetch - decode - execode)

→ Manage the program flow (make decisions)
* Zero * negative * overflow ...

مساحه أقل ، أعلى سرعه ، أعلى سعر ←



أكبر مساحه ، أقل سرعه ، أقل سعر ←

→ Transfer Data between itself and memory

processor

الشغال → Multi-processor ⟵ processor ⟶ Multi-Core

Buses: group of Common wires connect between parts of device.

1 Address Bus: ⟶

K-bit ⟶ 2^K devices.

2 Data Bus: ⟷

← كلما زاد كلما زاد معدل نقل البيانات من وإلى الـ MP

devices ⟷ MP

3 Control Bus: ⟷

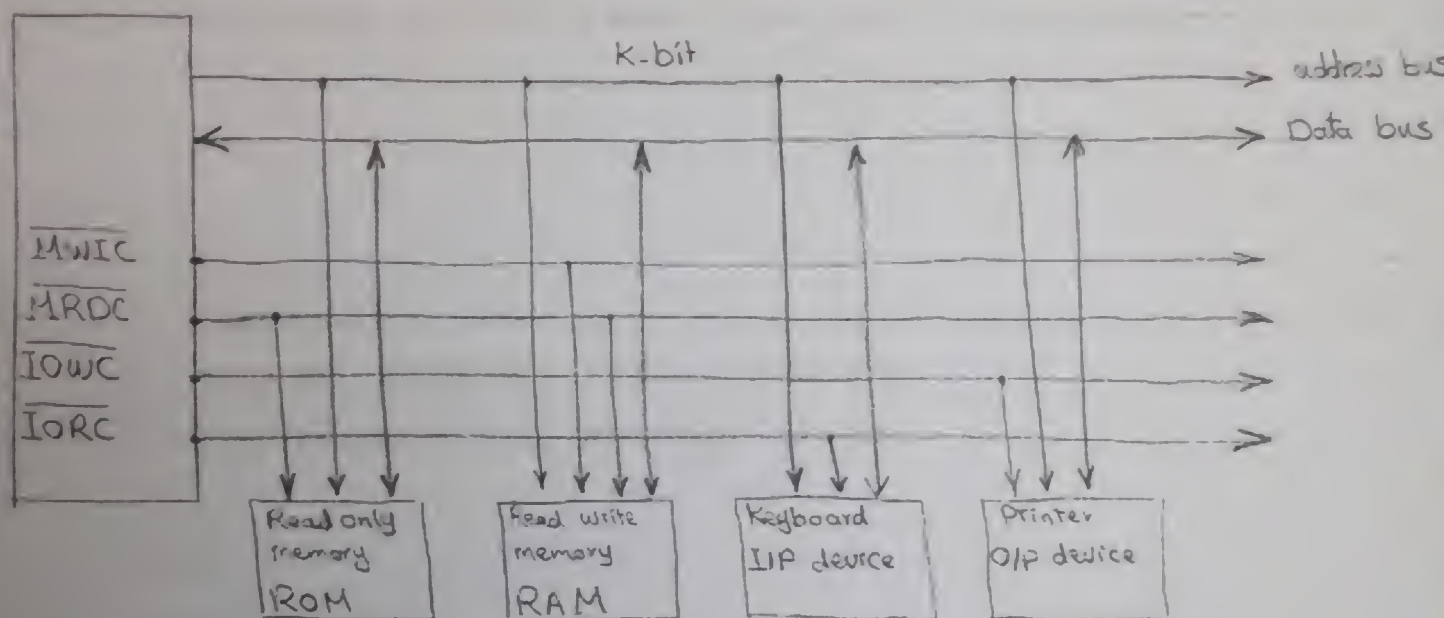
← ينقل Control Signal

كلما زاد كلما كان أفضل

devices ⟷ MP

←

Draw the block diagram of acomputer system showing the address . data and Control bus Signal ?



* Simple Control Signal

MWIC memory write Control Signal

MRDC memory read Control signal

IOWC I/O write Control Signal

IORC I/O read Control Signal.

→ MP → device التفاعل بين

1- MP → send address

2- Control Signal MRDC

3- Data transfer.

The Memory System is divided into "3 main points"

1- Transient program area (TPA)

2- System Area

3- extended memory system (xms)

$\boxed{I}$

→ holds

→ The length of TPA is 640 KB

→ programs → Interrupt vector

Ch2 : The Microprocessor & Its Architecture

8 bit	16 bit	8 bit	
AH	AX	AL	Accumulator
BH	BX	BL	Base Index
CH	CX	CL	Count
DH	DX	DL	Data
	SP		Stack Pointer
	BP		Base Pointer
	DI		Destination index
	SI		source index

IP
FLAG

CS	Code Segement
DS	Data Segement
ES	Extra Segement
SS	Stack Segement
FS	
GS	

special purpose.

Programming Model Of The Intel 8086 Pentium 4

32 bit		8bit	16bit	8bit	
EAX		AH	AX	AL	Accumulator
EBX		BH	BX	BL	Base Index
ECX		CH	CX	CL	Count
EDX		DH	DX	DL	Data
ESP			SP		Stack Pointer
EBP			BP		Base Pointer
EDI			DI		Destination Index
ESI			SI		Source Index

	IP	Instruction Pointer
	FLAGS	Flags

CS	Code
DS	Data
ES	Extra
SS	Stack
FS	
GS	

The Programming Model Of The Intel 80386 Pentium 4

كل 4 bit في الثنائي يعطي 1 bit في السادس عشر ←

20 bit ⟶ (XXXXX) Hexa.

Real Mode

Segement	Off Set	Special Purpose
CS	IP _ EIP	Instruction address
SS	① Then ② SP or BP , ESP / EBP	Stack address
DS	BX . DI . SI 8 bit 16 bit } numbers	Data address
ES	DI , EDI for String Instructions	String destination address
FS _ GS	No default	General address.

Programming Model

program Visible
- They are used during app - prog
- Ex: EAX, AX, AI, EBX ...
- → Mov AX, BX

program invisible
- not addressed directly during app - prog
- only 80286 _ P4 Contain this type
- ex: FIAG, EFIAG

SUB AX, BX بعد تنفيذ أمر

Purpose Of Use

General purpose
EAx , EBX
Hold Data address or off set +
البعد عن البدايه

Special purpose
CS, DS, ES, SS

Int X = 3
address data

Addressing modes (How the memory is accessed)

Addressing mode ← كثيره لكثره طرق تنظيم البيانات داخل الداكره
وظيفتها الوصول إلى الداكره

1- Real mode 1 MB only
2- Protected mode upto 4GB

« Real mode »

→ Accumulator EAx . AX . AL . AH

Can be used as 32bit → EAX , 16 → AX , 8bit → AH, AL

[I] Special purpose :

← طرف أساسى فى عمليات الضرب والقسمة

→ Div Ax , BX : $\frac{Ax}{Bx}$

Ax : يوضع به الجزء الصحيح
DX : (Data register) يوضع به باقى القسمه "الكسر"

→ Mux $\overset{16}{Ax}$, $\overset{16}{BX}$ → Mux EAX , EBX
16 * 16 → 32 bit 32 32

DX	AX	EDX	EAX
high bytes	low bytes	high bytes	low bytes

Base Index

Can be used as 32 bit → EBX , 16 bit → BX

8 bit → BH , BL

General purpose → hold off set for Data Segement

Note : <u>In Real Mode</u>

we have 64 KB processor , 5 bit memory

→ To access the memory للوصول للذاكرة

← processor من الـ

$64 K = 2^6 \cdot 2^{10} = 12^{16}$

$(16\ bit)_B \rightarrow (4\ bit)_{Hexa.}$

→ from 4 bit processor how to reach 5 bit memory

we add zero from the left side to Base

→ to reach to any address in the memory we need Base and off set

البداية Base is in Code Segement

∴ Current address$_{(real)}$ = $(CS)_0$ + off set

Ex: CS = 4000 Hex , IP = 2342

Solution

current address = $(CS)_0 + IP = 40000 + 2342 = 42342$ Hex

Count

To Count in any process with loop we need big register

loop CX , ECX

Repetition CX

shift CL we need small number of loops

Data

Can be used as 32 bit → EDX   16 bit → DX

8 bit → DH , DL

General purpose → Div , Mul   الضرب والقسمة

Stack pointer

Can be used as 16 bit → SP

hold the off set of Stack segement (SS) ←   الأولوية له

Base pointer

Can be used as 16 bit → BP

used as off set for Stack segement (SS)

Destination Index

Can be used as 16 bit → DI   له الأولوية بعد BX

hold the off set of Data segement

Source Index

Can be used as 16 bit → SI   له الأولوية بعد DI

hold the off set of Data segement

Instruction Pointer

Can be used as 16 bit → IP

hold the off set of current instruct to be excuted (in Code Segement)

Flag invisible

Over		S	Z		P	C
	...			...		

Can be used as 16 bit - 32 bit

→ Help microprocessor to take the corect way

1 Over flow happened when → +ve + +ve = -ve
→ -ve + -ve = +ve

2 Carry (C)

```
        11111010
Carry   1110111 0 +
  1  ←  1101000 0
```

3 Parity (P)

1 → number of one's is even

0 → number of one's is odd

نستفيد من هذا عندما يتم نقل Data من مكان لآخر نتأكد من أن
Data تم نقلها بدون فقد حيث اذا كان عدد 1's زوجي بعد النقل وكذلك P=1
كان النقل تم بالصوره المطلوبه والعكس يكون فيه فقد في Data

Ex: SS = 3245 H

BP = 2342

Sol

Base address = 33450 H

Current address = 33450 + 2342 = 35792

Ending address = Starting + FFFF

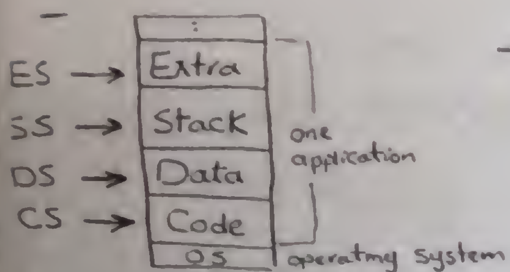
= 33450 + FFFF

Processor Mode

Processor $\xrightarrow[\text{2 modes}]{\text{access}}$ Memory

[1] Real Mode

- access 1^{st} MB memory
- run Only one application
- DOS
-



→ Every part of 64 K byte

Memory

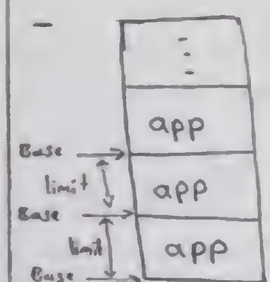
→ How to access memory?

- $EA = (Base)_0 + offset$

Base (starting)	& Offset
DS	→ Bx, DI, SI
SS	→ SP, BP
ES	→ DI
CS	→ IP

[2] Protected Mode. (80286 and above)

- access up to 4GB
- run more one application.
- GUT windows, linux
-



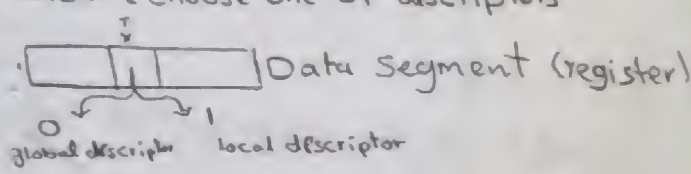
→ every application has different size from the other

Memory

up to 4GB

→ How to access memory?

1 – Selector : Choose One of descriptors



2 – Descriptor

- each descriptor 8 byte.
- There are 8192 descriptor for every application
- → local Descriptors : describe local applications
  → Global Descriptors : describe global applications.
- Knows us ① Base : application start in memory
  ② limit : the depth of the end from base
  → end = Base + limit
  ③ access right → a- Code
  b- data: Read only / Read / Write
  c- periority

Application

Global (system) Application
→ The data of this type is public.

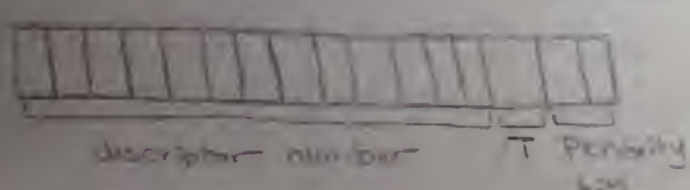
Local (user) Application.
— The data of this type is private.

TF

Descriptors

Global Descriptor	Local Descriptor.
— describe Global applications	— describe local applications.
— 8192 descriptor (number)	— 8192 descriptor (number)
— each descriptor 8 byte (size of each one in memory)	— each descriptor 8 byte (size of each one in memory)
— Total size $= \frac{8\,byte * 8192}{1024} = 64\,KB$	— Total size $= \frac{8\,byte * 8192}{1024} = 64\,KB$
— each descriptor describe • Base • limit • access right	— each descriptor describe • Base • limit • access right

الـ processor فيه عدد من registers لـ يتعامل مع الذاكرة
وبالتالي تكون الاختلاف بين الـ real mode و protected mode
في طريقة تفسير محتوى الـ registers حيث في الـ real mode
$EA = (Base \cdot 16) + offset$ و في الـ protected mode كالتالي



① Periority bits

0	0	← high periority
0	1	
1	0	
1	1	← low periority

← if the processor works with any program with 00 periority and get an interrupt from another program with 01 periority the processor will still work with the first program but if the 1st program with 11 periority and get an interrupt the

We Know ① the type of descriptor (global - local)
② the number of descriptor (one of 0 to 8191)
③ Periority bits

From Register

We Can Know ① Base ② limit ③ Access right

From Descriptor ← which we get from register.

السؤال : أين تقع جداول descriptor ؟

① لو في processor :

② لو في Memory : كيف ولماذا نخدم الوصول الى الذاكرة ومن ان هناك access الذاكرة مرتين

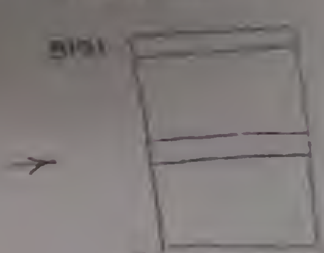
③ Cash Memory : ذاكرة بين processor و main Memory لتسريع عمل processor وسعتها صغيرة توضع فيها البيانات المكررة ولذلك هنا هي الحل الأمثل حيث يتم وضع الاجزاء المتكررة

Example:

CS = 2075 H

0	0	1	0	0	0	0	0	0	1	1	1	0	1	0	1

T = 1 local descriptor



→ Cash Memory → level 1 : in processor

→ level 2 : between memory and processor

The Descriptor Formats For 80286 P4 microprocessor

7	00000000	00000000	6
5	Access rights	Base (B23–B16)	4
3	Base (B15–B0)		2
1	limit (L15–L0)		0

The Descriptor Formats For 80386/80486/P/P pro/P II microp–

7	Base (B31–B24)	G	D	0	AV	limit (L19–L16)	6
5	Access rights	Base (B23–B16)					4
3	Base (B15–B0)						2
1	limit (L15–L0)						0

Ch3 Addressing modes (Real Model)

		Examples
[1] Immediate Addressing Mode	I → R ↑ information register	Mov BL, 44 , Mov BH, 44 H Mov CL, 1001110 B Mov ECx, FC23F34EH
[2] Register Addressing Mode.	R ⟷ R ↑ register register	Mov CL, BL (✓) Mov Cx, BX (✓) Mov EAx, EBX (✓) Mov Ds, Cx (✓) Mov Ds, Cs (x) Mov Cs, BX (x) Mov BX, CS (✓)

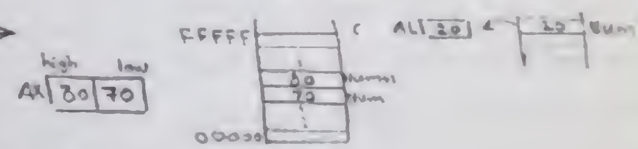
[3] Direct Addressing Mode
Direct addressing
↓ R: EAX, AX, AH, AL
Displacement (off set) addressing
↳ Any Register.
	R ⟷ M
	memory ↑
		Direct
		Mov AL, NUm
		Mov Ax, NUm
		Mov Num, Ax
		Mov EAX, Num
		Displacement
		Mov CL, Num
		Mov Cx, Num
		Mov EBx, Num
		Mov Dx, [1023]

[4] Register Indirect Addressing Mode[ ]
↳ (BP, BX, SI, DI)
	R ⟷ [M]	Mov Ax, [Bx]
		Mov [BP], DL
		Mov EDX, [DI]
		Mov [BX], [SP] (X)

[5] Bass_plus_Index Addressable Mode EBX → BX EBP → BP ESI → SI EDI → DI use with one dimention array	R ⟷ M	Mov Dx, [BX+DI] Mov WORD / DWORD PTR [BX+DI], DX

[6] Register_Relative Addressable_Mode
Register → BP, BX, SI, DI / EBP, EBX, ESI, EDI
رقم off set
مكان فى الذاكره
	R ⟷ M	Mov Ax, [BX+1000]
		Mov Ax, 1000[DI]
		Mov WORD / DWORD PTR [BX+1000], AX

[7] Base_relative plus Index Addressable Mode Used in two dimention Array	R ⟷ M	Mov WORD PTR 1000[BX+DI], Mov DH, [BX+DI+20 H]

Effective Address (EA)	Notes
No E.A	( )$_{10}$ عشرى ( )$_{B}$ ثنائى Binary ( )$_{H}$ السادس عشر , Characters
R_{source} (Rs) الذى نحصل منه على البيانات Register	R ⟷ R has the same size 8 ⟷ 8, 16 ⟷ 16 ← لا يجوز بأى حال من الأحوال تغير Code segment (CS) الذى يحتوى على عنوان الأوامر داخل الذاكره ولذلك لا يجوز النقل من segement الى آخر
E.A = memory = Num → EA هو المكان الذى نضع به أو نأخذ منه البيانات ولا يتأثر بنوع Instruction أو يعنى أصح لا يتأثر به	1→ Memory ⇝ Byte addressable مترقمه Byte by Byte 2→  little endian assignement
$EA = (seg)_{+0} + $ off set ① $EA = (DS)_0 + BX$ ② $EA = (SS)_0 + BP$ ③ $EA = (DS)_0 + DI$	• عند نقل بيانات من Register الى Memory لابد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable Mov [BX], AL ← يتم تحديد مساحه التخزين ← يتم التعديل الى Mov Byte PTR [BX], AL Mov WORD PTR [BX], AL Mov DWORD PTR [BX], AL
$EA = (DS)_0 + BX + DI$ starting address	يظل effective address ثابت
$EA = (DS)_0 + BX + 1000$	
$EA = (DS)_0 + \begin{matrix} BX \\ DI \\ 1000 \end{matrix}$	عند نقل بيانات من Memory الى Register لا نحدد المساحه للتخزين ولا توجد مشاكل

Ch3 Addressing modes (Real Mode)

		Examples
[1] Immediate Addressing Mode	I → R I ↑ information R ↑ register	Mov BL, 44 , Mov BH, 44 H Mov CL, 1001110 B Mov ECx, FC23F34EH
[2] Register Addressing Mode.	R ↔ R ↑ register ↑ register	Mov CL, BL (✓) Mov Cx, BX (✓) Mov EAX, EBX (✓) Mov DS, CX (✓) Mov DS, CS (x) Mov CS, BX (x) Mov BX, CS (✓)

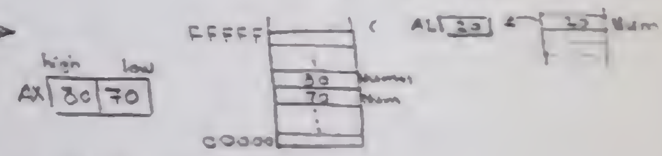
[3] Direct Addressing Mode
Direct addressing
↓ R: EAX, AX, AH, AL
Displacement (off set) addressing
↳ Any Register.
	R ↔ M
	memory ↑
		Direct
		Mov AL, NUm
		Mov Ax, NUm
		Mov Num, Ax
		Mov EAX, Num
		Displacement
		Mov CL, Num
		Mov Cx, Num
		Mov EBX, Num
		Mov DX, [1023]

[4] Register Indirect Addressing Mode[]
↳ (BP , BX , SI , DI)
	R ↔ [M]	Mov Ax, [Bx]
		Mov [BP], DL
		Mov EDX, [DI]
		Mov [BX], [SP] (X)

[5] Bass_plus_Index Addressable Mode EBX → BX EBP → BP ESI → SI EDI → DI use with one dimention array	R ↔ M	Mov DX, [BX+DI] Mov WORD / DWORD PTR [BX+DI], DX

[6] Register_Relative Addressable_Mode.
Register → BP, BX, SI, DI / EBP, EBX, ESI, EDI
Relative → رقم off set → مكان في الذاكرة
	R ↔ M	Mov Ax, [BX+1000]
		Mov Ax, 1000[DI]
		Mov WORD / DWORD PTR [BX+1000], AX

[7] Base_relative plus Index Addressable Mode Used in Two dimention Array	R ↔ M	Mov WORD PTR 1000[BX+DI], A Mov DH, [BX+DI+20H]

Effective Address (EA)	Notes
NO EA	( )$_{10}$ عشرى ( )$_{B}$ Binary ثنائى ( )$_{H}$ السادس عشر , ' ' Characters
R_{source} (R_s) الذى نحصل منه على البيانات Register	$R \leftrightarrow R$ has the same size 8 ↔ 8, 16 ↔ 16 ← لا يجوز بأى حال من الأحوال تغير Code segment (CS) الذى يحتوى على عنوان الأوامر داخل الذاكره وكذلك لا يجوز النقل من segement الى آخر
E.A = memory = Num ⇒ EA هو المكان الذى نضع به أو نأخذ منه البيانات ولا يتأثر بنوع Instruction أو بمعنى أصح لا يتأثر بحجمه	1→ Memory ⇝ Byte addressable مترقمه Byte by Byte 2→  little endian assignement
EA = (seg)$_{+0}$ + off set ① EA = (DS)$_0$ + BX ② EA = (SS)$_0$ + BP ③ EA = (DS)$_0$ + DI	← عند نقل بيانات من Register الى Memory لا بد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable Mov [BX], AL ← لم يتم تحديد مساحه التخزين ← لذا يتم التعديل الى
EA = (DS)$_0$ + BX + DI starting address	Mov Byte PTR [BX], AL Mov WORD PTR [BX], AL Mov DWORD PTR [BX], AL
EA = (DS)$_0$ + BX + 1000	ويظل effective address ثابت
EA = (DS)$_0$ + BX / DI / 1000	← عند نقل بيانات من Memory الى Register لا تحدد المساحه للتخزين ولا توجد مشاكل

Ch 4 : Data Movement Instructions

1- Assembler & Deassembler

2- Stack (push. pop)

3 - Instructions

——————————

Assembler & Deassembler

Hint : Assembler : transform the assembly Code to machine Cod

Deassembler : transform the machine Code to assembly Code

→ machine Code (0's , 1's)

⇝ Modes of Instruction

[1] 16 bit Instruction mode (8086)

Opcode D W	Mod REG R/M	Displacement	Immediate
← 1-2 bytes →	← 0-1 bytes →	← 0-1 bytes →	← 0-2 bytes →

[2] 32-bit Instruction mode (80386 above)

Address Size	Operand Size	OpCode D W	Mod REG R/M	Displacement	Immediate
← 0-1 bytes →	← 0-1 bytes →	← 1-2 bytes →	← 0-1 bytes →	← 0-4 bytes →	← 0-4 bytes →

To transfere from assembler to deassembler follow this steps

[1]

Operand Size	Address Size	mode
66 H	67 H	
×	×	– 16-bit instruction mode with its default (8 – 16 bit) registers ($8 \uparrow w=0$, $16 \uparrow w=1$)
✓	×	– 16 bit instruction but works with (8 – 32 bit) registers. ($8 \uparrow w=0$, $32 \uparrow w=1$)
✓	✓	– 32 bit instruction works with its default (8 – 32 bit) register ($8 \uparrow w=0$, $32 \uparrow w=1$)
×	✓	– 32 bit Instruction but works with (8 - 16 bit registers. ($8 \uparrow w=0$, $16 \uparrow w=1$)

2 | Opcode | D | W |

→ Opcode: has machine Code of the instruction (Mov , sto , swop)
we get this Code from the table
like Mov ⟶ 100010

→ D → D=1 (when we transfer from Memory to register)
→ D=0 (when we transfer from register to Memory)

→ W → W=1 (word / Dword)
→ W=0 (Byte)

3 | MOD | REG | R/M |

→ MOD : Its machine Code can be get from this table

MOD	Function		Ex:
00	No Displacement	R/M is memory	Mov AL, [DI]
01	8-bit Sign-extended displacement		Mov AL, [DI+2]
10	16-bit displacement		Mov AL, [DI + 1000]
11	R/M is a register		

→ REG : Its machine Code can be get from this table

Code	W=0 (Byte)	W=1 (word)	W=1 (DWord)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
	BH	DI	EDI

Code	Seg
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

→ R/M : to get the code of R/M we will first we want to know if R/M is Memory or Register that depend on MOD

IF MOD = 11 ∴ R/M is registers
we can get its code from register table

code	W=0 (Byte)	W=1 (word)	W=1 (Dword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

IF MOD = (00, 01, 10) ∴ R/M is Memory
we can get its code
from this table

R/M code	Addressing mode	Function
000	DS : [BX+SI]	DS: [EAX]
001	DS : [BX+DI]	DS: [ECX]
010	SS : [BP+SI]	DS: [EDX]
011	SS : [BP+DI]	DS: [EBX]
100	DS : [SI]	uses scaled-index byte
101	DS : [DI]	SS: [EBP]*
110	SS : [BP]* special addressing mode	DS: [ESI]
111	DS : [BP]	DS: [EDI]
	16-bit R/M memory addressing mode	32-bit addressing mode selected by R/M

4 Displacement

(1) 1 byte ☐☐ → Put as it توضع كما هي

ex: 1 → 01 , 21 → 21

(2) 2 byte ☐☐ ☐☐ → Swap high, low

ex : 1234 → 3412

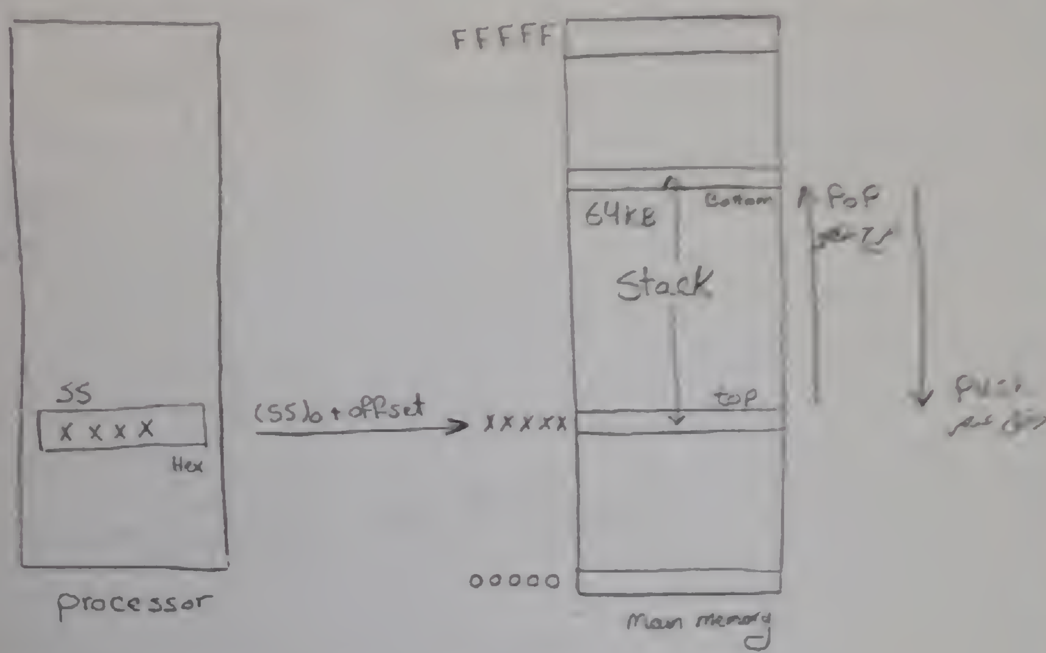
ex : 123 → 2301

Stack
(pop & push)

→ The PUSH and POP instructions are important instructions that store and retrieve data from the LIFO (last In First Out) stack memory

Stack (LIFO)
↓
two operations

- Pop
  يخرج عنصر
  ويحرك
  pointer
- Push
  يحرك pointer
  ويدخل عنصر

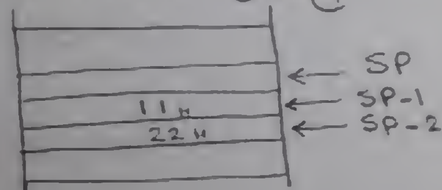


عندملأ Stack بالعناصر ووضع عنصر آخر بها يحدث
Cycle nature
اى يتم حذف ما بالعنوان الاول ووضع العنصر الجديد فى مكانه

→ Push AX (✓)

ضع محتوى AX فى stack

AX | 11_H | 22_H |
high byte, low byte



→ Push 5262_H (✓)

→ Push [BX] (X) ← هنا لم يحدد المساحة التى يخزن بها البيانات فى memory

EA = (DS)o + BX

The Correct
Push Byte PTR [BX] (✓)
Push (word - Dword) PTR [BX] (✓)

→ Push CS (✓) → هنا لم يغير محتوى CS

* **Hint**

22_H can be put in memory as

1. Byte $\boxed{22_H}$
2. word $\boxed{00}$ $\boxed{22}$
3. Dword $\boxed{00}$ $\boxed{00}$ $\boxed{22}$

... is performed on bytes, words or doublewords,

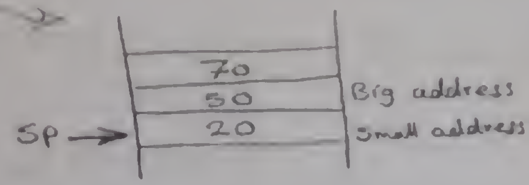
[illegible]

Pop ← stack من data

Pop 2345 H (X)

Pop CS (X) → we mustn't Change CS

Pop BX (✓) →



1) BX | 50 | 20 |
high low

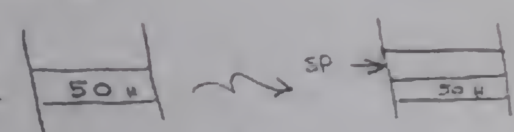
high address → high byte

low address → low byte

2) Then SP = SP + 2

Pop Byte PTR [BX]

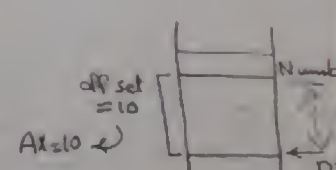
EA = (DS)₀ + BX →



Instructions

- LEA
  - LDS, LSS, LFS, LGS, LES
  - * off set
- LODS, StoS, MovS, INS, Outs.
  - * IN, Out
  - * X ChG
  - * BSwap
  - * CMOS

→ [I] LEA (load effective address)



ex: LEA AX, Numb ≡ Mov AX, off set Numb

هذا الأمر معناه وضع بعد المكان Numb من البدايه في AX

ex: LEA BX, [DI] → EA = (DS)₀ + DI ≠ Mov BX, offset [DI] (X)

→ LDS

ex: LDS BX, [DI]

10
70
60
50

$EA = (DS)_0 + DI = xxxxx$ →

BX | 60 | 50 |

DS | 10 | 70 |

← يتم تحديد بداية العنوان من EA

← نأخذ من بايتات memory حجمها بحجم المكان التى سوف توضع به

Hint

في المثال BX حجمه 16 bit اى 4 byte ∴ نأخذ مكانين من الذاكرة ونضعهم في BX بحيث يوضع العنوان الأول من الذاكرة في low byte والمكان الثاني في high byte

$byte = 2^{bit}$

memory size

← ثم نأخذ العناوين التاليين ونضعهم في DS حيث أن الأمر LDS

→ LSS

ex: lss EAX, [BX + DI]

15
14
13
12
11
10

$EA = (DS)_0 + BX + DI = xxxxx$ →

EAX | 13 | 12 | 11 | 10 | where EAX → 32 bit

SS | 15 | 14 |

→ LES

ex: lES BX, [DI]

80
70
60
50

$EA = (DS)_0 + DI = xxxxx$ →

BX | 60 | 50 |, ES | 80 | 70 |

→ LFS

ex: LFS BX, [DI]

70
50
60
80

$EA = (DS)_0 + DI = xxxxx$ →

BX | 60 | 80 |, FS | 70 | 50 |

→ LGS

ex: LGS BX, [DI]

4
3
2
1

$EA = (DS)_0 + DI = xxxxx$ →

BX | 2 | 1 |, GS | 4 | 3 |

LCS EAX, [DI] (X)

because CS mustn't be Changed.

String Instructions

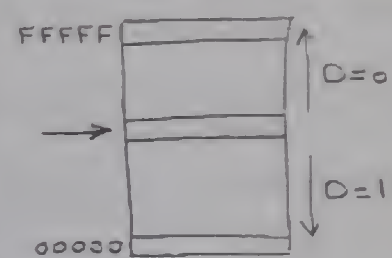
LODS, STOS, MOVS, INS, OUTS

This instructions depend on direction F.

| | D | | FLAG

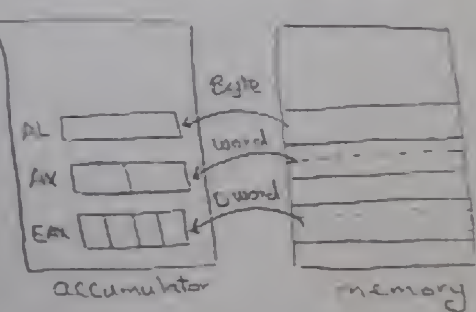
D = 0 → autoincrement mode
نسير في اتجاه زيادة العنوان

D = 1 → autodecrement mode
نسير في اتجاه نقص العنوان



I LODS

نأخذ من memory ونضع في accumulator ثم ازود offset أو اقلله حسب D



LODSB → byte
LODSW → word
LODSD → double word
المكان غير معروف مباشرة يوصله عن طريق EA

LODS list → byte (DB)
LODS Data → word (DW)
LODS FROG → double word (DD)
المكان الذي تم وضع فيه في الذاكرة معروف مباشرة

list : DB 20H | 20 | list

list2 : DW 5070H | 70 | 50 | list2

list3 : DD 50702011H | 50 | 70 | 20 | 11 | list3

and Can be Signed integer (IMUL) or Unsigned

from [Source] --to--> [Destination]

[Data Segment] [Extra Segment]

offset SI DI

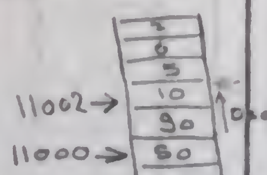
<u>ex:</u> DS=1000H, SI=1000H, D=0
execute LODSW then LODSD

<u>Solution</u>

$EA = (DS)_0 + SI = 10000 + 1000 = 11000$

$SI_{new} = SI_{old} + 2\,bit = 1002$

AX [90 | 50]



To execute LODSD

SI=1002, DS=1000, D=0

EA=10000+1002=11002

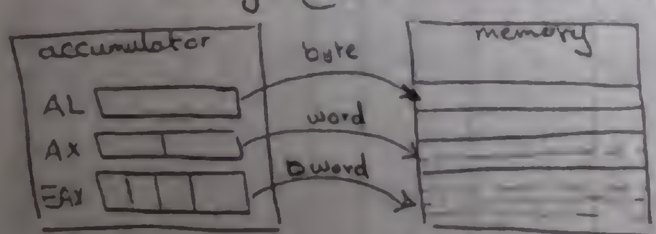
EAX [7 | 6 | 5 | 10]

$SI_{new} = SI_{old} + 4\,bit = 1002+4=1006$

If D=1 $SI_{new} = SI_{old} -$ 1 → byte, 2 → word, 4 → Dword

2 STOS

يأخذ من accumulator ويضع في memory



D ثم يزود offset او يقلله حسب

STOSB, STOSW, STOSD
STOS list (DB), STOS list2 (DW)
STOS list2 (DD)

3 MOVZX

يزود باقي خانات Register
بـ اصفار Zeros

MOVSX يزود باقي المساحة (Memory
بنفس الصحيح بالإشارة

4 XCHG

ابدل الاماكن

→ Both places must have the same SIZ

→ CS mustn't be Change

<u>Examples:</u>

XCHG AL, CL (✓)
XCHG CX, BP (✓)
XCHG Data, AX (✓)
XCHG DS, ES (X)
XCHG CS, AX (X)

5 BSwap

→ work only with 32 bit registers (80486 → P3, P4)

EAX [50 | 20 | 30 | 40] before

EAX [40 | 30 | 20 | 50] after

6 CMoV (Conditional Mov)

نقل مشروط

CMOVZ → هذا الأمر معناه
١- يرى خانة Z في Flag
Z=0 لا يفعل شيء
Z=1 ينقل

ex: CMOVZ AX, BX
if Z=1 ∴ AX=BX
else Do nothing

→ CMOVC → Chick Carry
if C=1 → Move
else Do nothing

→ CMOVNC → Chick No carry
if C=0 → Move
else Do nothing

Ch 5 Arithematic & Logic Instruction

1 Arithematic

Binary operator:
- ADD
- ADC
- SUB
- SBB

Unary operator:
- INC
- DEC

- MUL
- IMUL
- DIV
- IDIV
- CMP
- JA, JB, JAE
- JBE, JE, JNE

2 Logic

- AND
- OR
- NOT
- NEGATE
- TEST
- XOR
- BT
- BTC
- BTS
- BTR
- Shift
  - (logical) unsignal numbers → SHL (*2), SHR (/2)
  - (Arithematic) signed numbers → SAL, SAR
- Rotate
  - without Carry → ROR, ROL
  - with Carry → RCR, RCL

→ Arithematic

① ADD

→ ADD AL, 20 H → $AL_{new} = AL_{old} + 20$ [Immediate Addressing mode] EA = No E.A

→ ADD AL, BH → $AL_{new} = AL_{old} + BH$ [Register Addressing mode] E.A = BH

→ ADD AL, NUM → $AL_{new} = AL_{old} + NUM$ [Direct Addressing mode] EA = NUM

→ ADD AL, [BX] → $AL_{new} = AL_{old} +$ place in memory (1 byte) [Register Indirect Addressing Mode]

EA of [BX] = $(DS)_0 + BX$ →

→ ADD AX, [BX + 2000] → $AX_{new} = AX_{old} +$ place in memory (2 byte) [Register - Relative Addressing Mode]

EA = $(DS)_0 + BX + 2000$

→ ADD EAX, [BP + SI] → $EAX_{new} = EAX_{old} +$ place in memory (4 byte) [Bass plus Index Addressing Mod

EA = $(SS)_0 + BP + SI$

→ ADD AX, 2000 [BX + SI] → $AX_{new} = AX_{old} +$ place in memory (2 byte) [Base relative plus Index A.M

EA = $(DS)_0 + BX + SI + 2000$

→ ADD 2000[BX+SI], EAX → place in memory (4 byte) = place in memory (4 byte) + EAX

→ EA = $(DS)_0 + BX + SI + 2000$

V.I

① ADD ES, DS (X)

② ADD [BX], [BP] (X)

③ ADD CS, AX (X)

④ ADD AX, CS (✓)

⑤ ADD CX, BL (X)

③ CMP (Compare) → we compare between minus to make ...

② ADC (add with Carry)

Carry → $\boxed{C}$

$$\begin{array}{r} Bx\ \ Bx \\ +\ Bx\ \ Cx \\ \hline Dx\ \ Ax \end{array}$$

→ ADC AL, 20H → $AL_{new} = AL_{old} + 20_H + C$

ويأخذ باقي أشكال Addressing Modes مثل الأمثله السابقه لـ ADD

~ ——————— ~

③ SUB

→ SUB AL, 20H → $AL_{new} = AL_{old} - 20_H$

وأيضا هو وباقي Instructions تأخذ نفس أشكال Addressing Modes مثل الأمثله السابقه لـ ADD

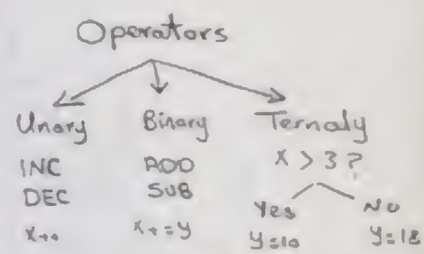
~ ——————— ~

④ SBB (sub with Borrow) Borrow : هي نفس خانة Carry في Flag

→ SBB AL, 20H → $AL_{new} = AL_{old} - 20_H - C$

~ ——————— ~

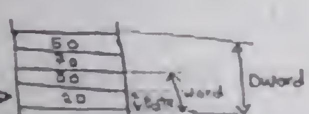
⑤ INC (Add 1)



→ INC AX → $Ax_{new} = Ax_{old} + 1$

V.I → INC Byte ptr / word ptr / Dword ptr [Bx]

$EA = (DS)_b + Bx$ →



→ Byte 20 + 1 = 21

→ word 80 20 + 1 = 8021

→ Dword 50 70 80 20 + 1 = 50708021

→ INC ES (X)

→ INC CS (X)

~ ——————— ~

⑥ DEC (sub 1)

→ DEC AX → $Ax_{new} = Ax_{old} - 1$

→ DEC ES (X)

→ DEC CS (X)

Multiplication is performed on bytes, words or doublewords, and can be signed integer (IMUL) or unsigned integer (MUL). The product after a multiplication is always a double-width product.

	Object / type	Examples:
(1) MUL [illegible] always in Accumulator [illegible]	ⓐ 8 bit Multiplication – The multiplicand is always in the AL register – The multiplier can be 8-bit register or any memory location – After the multiplication, the unsigned product is placed in AX, a double-width product	MUL BL MUL CL MUL Temp
	ⓑ 16 bit Multiplication – The multiplicand is always in the AX register – The multiplier can be 16-bit register or any memory location – After the multiplication, the unsigned product is placed in EAX, a double-width product.	MUL CX MUL word ptr [SI]
	ⓒ 32 bit Multiplication – Only the 80386 through P4 processors multiply 32 bit doubleword – The multiplicand is always in the EAX register – The multiplier can be 32-bit register or any memory location – The product (64 bit wide) is found in EDX-EAX where EAX contains the least-significant 32 bit of the product	MUL ECX MUL Dword ptr [ECX]
(2) IMUL The multiplicand is always in Accumulator – The product is in true binary form if positive, and in 2's comp if negative. – These are the same forms used to store all positive and negative signed numbers used by microprocessor	ⓐ 8 bit Multiplication – AL is multiplied by 8 bit register (ex. DH) or any memory location (ex. [BX]), the signed product is in AX	IMUL DH IMUL byte ptr [BX]
	ⓑ 16 bit Multiplication – AX is multiplied by 16 bit register (ex. DI), – The signed product is in DX - AX	IMUL DI
	ⓒ 32 bit Multiplication – EAX is multiplied by 32 bit register (ex. EDI) – The signed product is in EDX - EAX → Only the 80386 P4 processors multiply 32 bit double word.	IMUL EDI

(11) CMP

Division occurs on 8- or 16 bit numbers in the 8086 – 80286 proc and on 32 bit numbers in the 80386 P4. These numbers signed (IDIV) or unsigned (DIV) integers The dividend is alway a double-width dividend that is divided by the operand. There is no immediate division instruction any micro

	Object / type	Example
(9) DIV The Accumulator is used to store the dividend	(a) 8-bit Division – AX register to store the dividend – The dividend is divided by the Constents of any 8 bit register or memory location. – The quotient moves into AL after the division with AH containing a whole number remainder.	DIV CL DIV Byte ptr [BP]
	(b) 16-bit Division – is similar to 8 bit division except that instead of dividing into AX The 16-bit number is divided into DX – AX, a 32 bit dividend	DIV CX DIV NUMB
	(c) 32 bit Division – The 64-bit contents of EDX - EAX are divided by the operand specified by the instruction, leaving a 32-bit quotient in EAX and a 32 bit remainder in EDX	DIV ECX DIV DATA2
(10) IDIV	(a) 8-bit Division – AX is divided by 8 bit register (ex BL), the signed quotient is in AL and the remainder is in AH.	IDIV BL
	(b) 16-bit Division – DX-AX is divided by 16-bit register (ex SI), the signed quotient is in AX and the remainder is in DX	IDIV SI
	(c) 32 bit Division – EDX-EAX is divided by the doubleword Constents of the data Segment memory location addressed by EDI, the signed quotient is in EAX and the remainder is in EAX	IDIV DWORD PTR [E

pr ino

⑪ CMP (Compare)

→ We Compare between things to make other operations depend on the relation (< = >) between the things.

→ CMP AL, BH

JA → A > B
JB → B > A
JAE → A ≥ B
JBE → B ≥ A
JE → A = B
JNE → A ≠ B

① AL − BH
- AL > BH → Result +ve then S = 0, Z = 0
- AL = BH → Result 0 then Z = 1
- AL < BH → Result −ve then S = 1, Z = 0

Note: This Operation Changes S and Z in Flag

```
→ CMP AL, BH
   JA  Lable 1
   JB  Lable 2
       .
       .  other instructions
       .
   Lable 1: ...
   Lable 2: ...
```

① AL − BH

a) Result +ve ⇝ AL > BH
- Jumb to Lable 1 to do its operations neglecting JB Lable 2 and the other instructions.

b) Result −ve ⇝ BH > AL
- Jumb to Lable 2 to do its operations, neglecting the other instructions.

c) if AL isnot > BH or BH > AL
- go to do other instructions.

→ CMP [AL], [BH] (X)
→ CMP CS, SS (X)
→ CMP CS, AX (✓)
→ CMP AX, AL (X)

* ———————— *

→ Logic

① AND

→ AND AL, 20H

assume AL = 10H

$$
\begin{array}{ll}
AL = 00010000 & \\
20_H = 00100000 & \text{and} \\
\hline
AL_{new} = 00000000 = 0_H &
\end{array}
$$

② OR

→ OR AL, 20H

AL = 10H

$$
\begin{array}{ll}
AL = 00010000 & \\
20_H = 00100000 & \text{or} \\
\hline
AL_{new} = 00110000 = 30_H &
\end{array}
$$

③ NOT (1's complement)

→ NOT Byte Ptr / word Ptr / Dword Ptr [BX]

1's Complement → 0 ⇝ 1, 1 ⇝ 0

④ NEGATE (2's Complement)

Note: 2's Complement

نمشي عن اليمين ننزل 0 وأول واحد 1 يقابلنا ننزله وبعده نقلب 0 الى 1 والعكس

ex:	111010	,	001
2's Complement:	000110	,	111

④ TEST

يختبر الوزن الموضعي (128) خانه يحدد رقمها

وبالتالي يليه امر آخر JNZ , JZ

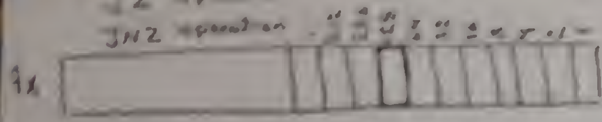
JZ → Jump if 0

JNZ → Jump if no Zero

→ TEST Ax , 128

JZ operation

JNZ operation



← result ?

* ——————— *

⑤ BT

يختبر bit واحده الموجود فقط في (80386-P4

→ BT Ax , 4

Test bit position 4 in Ax

* ——————— *

⑥ BTC → test and Complement

→ BTC Ax , 128

JZ op1

JNZ op2

← يقوم بعمل Test لـ bit واحده

ثم ① اذا كانت بـ 0 يحولها الى 1

② اذا كانت بـ 1 يحولها الى 0

* ——————— *

⑦ BTS → Test and Set

→ BTS Ax, 4

Test and set bit position 4 in Ax

← يقوم بعمل Test لـ bit واحده

ثم يضع لها 1

* ——————— *

⑧ BTR → Test and Reset

→ BTR Ax, 4

Test and Reset bit position 4 in Ax

← يقوم بعمل Test لـ bit واحده

ثم يضع لها 0

(1) Shift : Shift instructions position or move numbers to the left or right within a register or memory location.

*

1. Unsigned (logical) numbers

→ Shift Right SHR

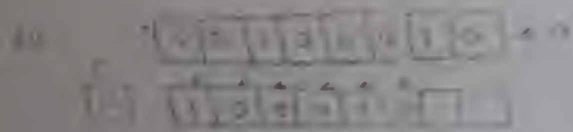
Ex SHR BX, 12 (BX is logically shifted right 12 places)

SHR ECX, 10 (ECX is logically shifted right 10 places)

→ Shift Left SHL

Ex: SHL AX, 2 (AX is logically shifted left 2 places)

SHL AL, 2 [illegible]



2. Signed (Arithmetic) numbers

→ Shift Right SAR ← في هذا النـوع لابد من الحفاظ على الإشاره ثابتـه

Ex: SAR SI, 2

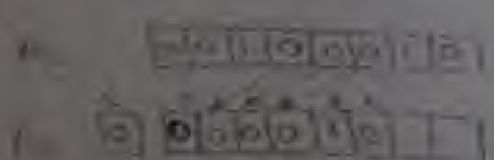
(SI is arithmetically shifted right 2 places)

SAR EDX, 14

(EDX is arithmetically shifted right 14 places)

→ Shift Left SAL

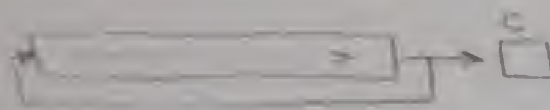
SAL AL, 2 [illegible] AL = 22 H



the information in a register or memory location, either from one end to another or through the Carry flag.

[1] Rotate without Carry

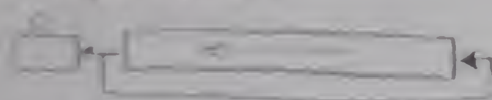
→ Rotate Right ROR



Ex: ROR WORD PTR [BP], 2

(The word contents of the stack segment memory location addressed by BP rotate right 2 places)

→ Rotate Left ROL



Ex: ROL SI, 14 (SI rotates left 14 places)

ROL ECX, 18 (ECX rotates left 18 places)

[2] Rotate With Carry

→ Rotate Right RCR

Ex: RCR AH, CL (AH rotates right through carry the number of places specified at CL)

→ Rotate Left RCL

Ex: RCL BL, 6 (BL rotates left through carry 6 places)